A Visual Analysis and Design Tool for Planning Software Reengineerings

Martin Beck, Jonas Trümper and Jürgen Döllner
{martin.beck}, {jonas.truemper}, {juergen.doellner}@hpi.uni-potsdam.de
Hasso-Plattner-Institute – University of Potsdam, Germany

***Abstract*—Reengineering complex software systems represents a non-trivial process. As a fundamental technique in software engineering, reengineering includes (a) reverse engineering the as-is system design, (b) identifying a set of transformations to the design, and (c) applying these transformations. While methods a) and c) are widely supported by existing tools, identifying possible transformations to improve architectural quality is not well supported and, therefore, becomes increasingly complex in aged and large software systems.**

In this paper we present a novel visual analysis and design tool to support software architects during reengineering tasks in identifying a given software's design and in visually planning quality-improving changes to its design. The tool eases estimating effort and change impact of a planned reengineering. A prototype implementation shows the proposed technique's feasibility. Three case studies conducted on industrial software systems demonstrate usage and scalability of our approach.

I. Introduction

Reengineering and *refactoring* are crucial engineering methods in software development and software maintenance processes. They are applied to an existing software system to alter its representation without changing its behavior [1]–[3]. The desired result is typically an improvement of a system's architectural and/or code quality, e.g., in terms of better comprehensibility and maintainability. Both methods are considered vital activities in particular for large systems. In fact, numerous agile software development methods include specific refactoring phases to ensure code quality [1], [4], [5].

Murphy-Hill and Black distinguish two types of refactoring using a dental metaphor [6]: *Floss* refactoring stands for small changes such as 'extract method' and is typically applied as drive-by action by developers. *Root-canal* refactoring (i.e., reengineering), in contrast, targets a larger set of changes, including major revisions of a software system's design, and is mostly planned by software architects. Studies show that floss refactoring is far more common than root-canal refactoring [7], [8]. Nevertheless, experience from industry has proven that numerous aged and grown software systems are exhibiting major design flaws [9], [10]. Although performing a root-canal refactoring is considered critical since meanwhile existing functionality may break, the long-term perspective of saving major expenses through less complex and better comprehensible design is reason enough for industry to take the risk [11], [12].

A typical root-canal refactoring scenario [13] includes the following steps (Fig. 1): (S1) the as-is *source design* is assessed and understood (*program comprehension, reverse engineering* [2]), (S2) potential problems in the source design, *design flaws*, are identified, (S3) the desired *target design* is planned (*forward engineering*), and (S4) a set of actions to transform the source design into the target design are determined. While S1 through S3 are rather straight-forward for small systems, they become increasingly difficult in larger systems: Deciding which possible target design will actually improve the code's comprehensibility and maintainability is considered a complex task [12], [14], [15].

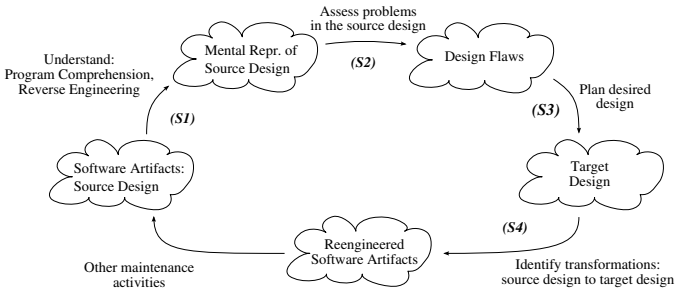


Fig. 1. Typical reengineering scenario within the software maintenance cycle: (S1) Understand the source design of a software system, (S2) assess design flaws therein and (S3) plan modifications to the design, and (S4) transform a system's artifacts from the source design into the desired target design. Finally, other maintenance activities involve modifications to the system's artifacts (and design) that potentially introduce new design flaws.

Existing refactoring tools and techniques [11], [16], [17] can, in general, cope with complex systems by providing abstractions and visual representations. However, they typically ignore that more complex refactoring tasks, in particular in terms of root-canal, require a significant amount of exploratory work beforehand for finding a proper transformation of a system's design. As a result, it is often impossible to determine reliable estimations of the required effort and the change impact of reengineering tasks, e.g., because of unknown dependencies.

The main contributions of this paper are (a) a novel concept for visual analysis and design that supports assessment of the as-is design (hierarchy and dependencies) and exploratory identification of potential transformations to it. By that, software architects are assisted in estimating effort and change impact of identified transformations. (b) We demonstrate the concept's feasibility by a prototype implementation. (c) The technique's use and scalability are shown by three case studies conducted on complex software systems.

978-1-4577-0823-7/11/$26.00@2011 IEEE

II. Related Work

Whereas most approaches target floss refactoring, only a few target root-canal refactoring. By contrast to the former ones, our approach aims at supporting root-canal refactoring.

A. Root-Canal Refactoring

Literature on root-canal refactoring can be separated into three categories: (1) Visual assessment tools focus on capturing and representing a system's design while (2) tools for quality assessment and (3) recommendation systems use a system design as input to compute their output.

Techniques of category (1) and (2) take a rather passive role in reengineering tasks in that they solely represent the as-is state of a software system. Our technique, in contrast, primarily targets a software architect's activities *after* assessing the as-is design.

1) Visual Assessment of Software Designs: Telea and Voinea [11] report results from a case study in which they analyzed the reasons for a software project not staying on time. Using bundled edges [18], they identified architecture violations. Although their approach does not support software architects in manipulating analyzed data, they applied it to the same domain as we do. Bourquin and Keller [12] present experiences from refactoring tasks in several industrial projects. Similar to our approach, they describe architecture violations as indicators on which subsequent reengineering can base on.

Lanza and Ducasse [16] project multiple metrics onto a system's structure or hierarchy using polymetric views. They argue that such combination of metrics can be used to identify refactoring opportunities. Wettel and Lanza [17] present a similar approach that projects metrics onto 3-dimensional treemaps. By that, design flaws measurable by metrics can be identified visually. Bohnet and Döllner [19] propose an extended, also treemap-based, approach that visually connects code metrics and a software system's evolution. Thereby, current and past development activities as well as design flaws can be assessed. Lewerentz and Noack [20] use clustered graphs with force-directed layouts to depict a software system's structure. Code metrics are mapped to visual properties of nodes and edges to support structure analysis and identification of code smells. Schwanke [21] introduces ARCH, a collection of tools for restructuring software. Since their visualization approach is based on graphs, it is inherently prone to scalability problems and is thus rather suited for floss-refactoring than for larger design adaptations.

2) Quality Assessment of Software Designs: Bengtsson and Bosch [22] use multiple quality attributes for assessing a software architecture's properties, such as modularity. They argue that software architects can iteratively improve an architecture using these quality attributes. Kang and Bieman [23] show how metrics for measuring design-level cohesion can be used to restructure software. While they propose a process template, it is rather coarse-grained and does not tackle the problem of identifying necessary changes to a system's design.

Marinescu [24] proposed detection strategies based on software metrics to identify design flaws in object-oriented systems. Ratiu et al. [25] propose an extension to Marinescu's approach that joins 'traditional' software metrics with code evolution data to classify code smells into 'good' or 'bad' ones. That is, code smells that are present for a long period of time and are touched frequently ('bad' ones), are more likely to be responsible for high maintenance effort and should be fixed prior to others. As orthogonal approach to ours, it could complement our technique by providing a ranking of the 'most expensive code smells'.

3) Recommendation Systems: Schwanke and Hanson [26] propose a tool for improving a software system's modularization using neural networks. The tool solely focuses on moving modules and does not consider other actions for improving modularization. In addition, scalability for larger systems is not evaluated. Hutchens and Basili [27] use clustering algorithms to recommend a partitioning of a subject software system into modules. A comparison of this recommendation with the given partitioning of a current design can then indicate potential problems therein. However, these recommendations have to be interpreted with care since selecting the optimal clustering algorithm depends on the scenario, and no rules for its selection are given. Choi and Scacchi [28] propose a reverse engineering technique that captures an existing system's design using a module intercommunication language. While their technique further provides recommendations for restructuring modules, it considers a system as a whole. Consequently, the technique does not support partial restructuring.

B. Floss Refactoring

The majority of floss-refactoring techniques focuses on fixing code smells mentioned in Fowler's book [1]. While this approach works well for code smells that are easy to fix, limitations of these tools become visible when more complex code smells of the book are concerned [6]. Hence, our approach could as well provide inspiration for improvements to floss refactoring tools.

1) Interactive Floss-Refactoring: Murphy-Hill and Black [29] present an add-in for the Eclipse IDE that provides users with an interactive indicator for specific code smells. In contrast to our approach, their tool does not include previews for change impacts of a refactoring. Müller and Klashinsky [30] introduce an approach for visual exploration and editing of source code as well as dependencies and structure of software systems. By contrast, this approach lacks sufficient scalability for large data sets.

2) Semi-Automatic Floss-Refactoring: Mayer et al. [31] propose a visual approach for supporting type-related refactoring (e.g., replace if-else by type hierarchy) in object-oriented systems. Similar to that, Tsantalis and Chatzigeorgiou [32] present a recommendation system as Eclipse add-in for identifying refactoring opportunities related to polymorphism. Contrary to our approach, both techniques rely on code smells as indicators for possible refactoring and thus lack scalability for root-canal refactoring.

Hayashi et al. [33] use a collection of previous user-interactions with an IDE to rank and recommend possible

refactoring activities related to these interactions. In contrast, this approach only works if previous user-interactions are available and thus does not work when applied to pre-existing software systems. Simon et al. [14] propose to use cohesion metrics to derive a geometric distance of two software artifacts. A visual representation of a software system based on measured distance values can then be used as indicator for refactoring opportunities. Their approach is orthogonal to ours and their distance measure could be used to further support root-canal refactoring by providing recommendations.

III. Visual Analysis and Design for Software Reengineering

When starting an explicit reengineering phase, architects typically have in mind one of two main tasks: Either they plan to improve a software system's overall code quality or they have a specific reengineering goal but with unknown feasibility and time consumption.

To improve the overall quality of a software system's architecture and design, potential flaws and issues have to be identified first. Typically, architects want to remove unnecessary complexity from a system, disentangle module dependencies or clarify responsibilities within its code. These goals are best practices and follow common sense [4], [13], [34].

On the other hand, architects may have in mind specific reengineering tasks such as "Remove all dependencies from module A to module B". Frequently, these tasks are a result of a previous architecture quality analysis. In general, as neither software architects nor developers have full system knowledge, such reengineering tasks may imply unknown risks. For example, a misestimation of required effort may lead to time consuming reengineering phases. During these reengineerings, developers may even notice that a specific planned reengineering is not feasible in practice. Therefore, it is important to check feasibility of reengineerings and required effort beforehand and, thereby, prevent excessive cost increase.

For both refactoring tasks, we propose a visual and interactive design tool to support architects. It consists of a visualization depicting a software system's structure with embedded inner relationships, as well as interaction metaphors enabling virtual manipulation of software entities. Architects can identify architecture violations, cluttered module dependencies or other design flaws. Visual manipulation enables them to plan alterations of a system's structure and to quickly see the results of planned modifications such as resolved indirect module dependencies. As the underlying source code is not changed, this concept facilitates experimentation and testing of reengineering hypotheses.

A. Data Model

To cover a broad range of distinct software systems, in particular aged and legacy systems, our concept abstracts from specific programming languages. Instead, a generic model of a software's structure is required to support multiple programming paradigms and intermixed software systems.

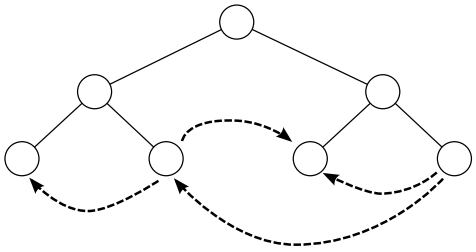


Fig. 2. Conceptual representation of both software hierarchy and dependency relations in one unified view. The software hierarchy graph consists of non-leaf nodes such as modules or packages and leaf nodes such as classes or files. Hence, solid edges indicate parental relations. Dashed arrows represent a uses-relation between two nodes.

Typically, software consists of hierarchically structured modules or packages; modules support a separation of concerns by grouping similar functionality into distinct software entities. Frequently, such a module hierarchy maps to a directory hierarchy. Within a software's modules, low-level software entities such as files or classes exist, which contain the actual source code. Between these software entities there are dependencies, e.g. a class referring to another class.

This concept of representing a software's modular hierarchy and dependencies between its entities as a compound directed graph is not limited to common programming languages such as C++ or Java. For example, an SAP ABAP software system or a legacy COBOL system consist of a similar modular hierarchical structure although different entity and dependency types are used.

B. Visual Representation

Our concept facilitates visual, interactive analysis and design of changes to a software's structure and its dependencies. To actually visualize a software's compound structure and dependency graph, a layout technique is required. Fig. 2 illustrates a small example software hierarchy and its dependencies on top using a rooted tree layout. Leaf nodes correspond to low-level software entities, e.g., files or classes, and non-leaf nodes represent architectural entities such as modules or packages. Directed connections between nodes indicate dependencies within this hierarchy. While our interaction concept is independent of a specific tree and compound graph visualization technique, it has four requirements to potential techniques:

1) Unified Structure and Dependency View: The visualization and its layout algorithm should provide a unified view of a software system's hierarchical structure and its inner module dependencies. Enhancing standard tree visualization techniques, e.g., the rooted-tree layout used in the example figures, by adding dependency edges on top, typically suffers from occlusion of nodes by edges when applied to larger tree structures. Thus, reduction of visual clutter has highest priority. Additionally, when layouting edges, the visualization should emphasize edges crossing parental module boundaries as these are more likely to be of interest than inner-module dependencies.

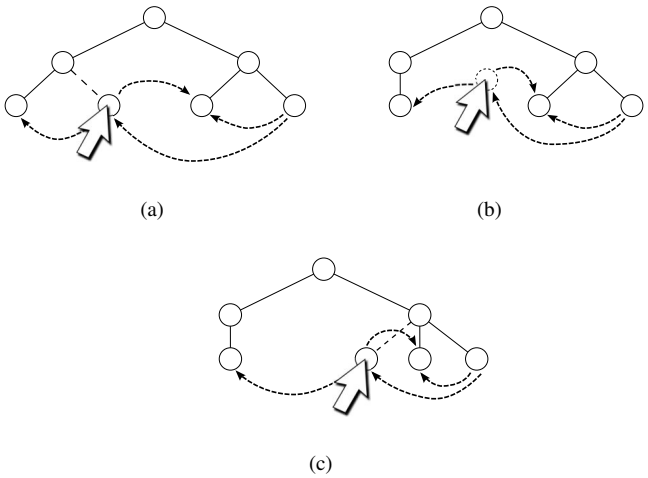


Fig. 3. Drag and drop concept. (a) When users start to drag a module, the representing node is detached from its parent and the layout recomputed. (b) While moving the node, its incoming and outgoing dependency edges change as well and stay connected to the detached module. (c) Dropping the module to a new place reinserts it into the hierarchy and, again, recomputes the layout.

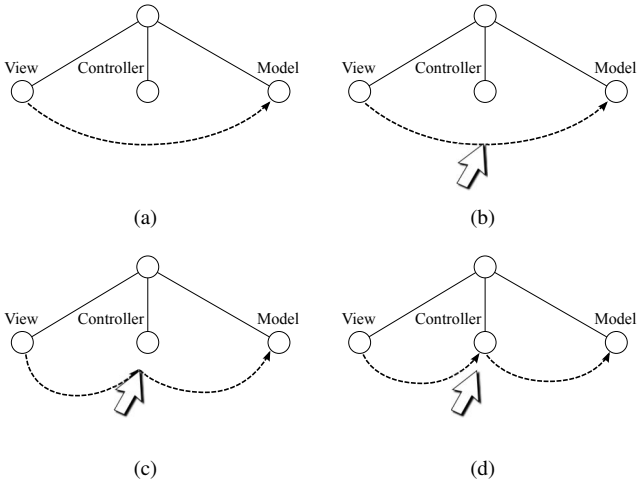


Fig. 4. Dependency splitting. (a) A dependency violates the intended model-view-controller architecture. (b) The dependency edge is grabbed in the middle. (c) When starting to drag the dependency edge, it is split into two separated edges. (d) Both new dependency edges are dropped onto the controller module.

2) Layout Stability: Inserting, moving and removing software modules as well as creating new dependencies should leave the layout almost unchanged. Any substantial realignment of visualization items upon an interaction leads to unnecessary user confusion. Hence, spatial proximity has to be obeyed during re-layouting when a software architecture is modified by the architect.

3) Scalability: Scalability of the visualization improves its applicability to large software systems. Software architects not only need to gain a high-level overview of their system, but also need to have a deep insight into detailed parts. Combining the ability to layout top-level structures, which aggregate low-level information, with zooming is an essential part of our visualization concept.

4) Performance: Finally, the visualization technique should efficiently support interactivity. For example, real-time layout recomputation upon module rearrangement is required to give reasonable feedback to users.

C. Interaction Techniques

To visually analyze and design software hierarchies, system architects require a broad set of interaction techniques. In our concept, several search and filter mechanisms enable to view distinct aspects of a software architecture and to quickly find specific modules. Searching for text fragments highlights relevant modules in the visualization and clicking on them reveals further detailed information. A drill-down feature allows users to restrict the visualization to a submodule and its descendants. Only packages and dependencies within this submodule are displayed. Additionally, modules can be collapsed to hide their substructures. Dependency edges will be aggregated upwards to the collapsed module.

For interactive discovery of architecture violations, our concept provides three dependency visualization modes. In the first mode, all dependencies are shown. The second mode shows only dependencies related to specific modules, i.e., their incoming and outgoing dependencies. By hovering and clicking, users select modules of interest. When a top-level module is selected, dependencies of its submodules are shown. Modules targeted by the relations are highlighted and their labels are enlarged. A third dependency interaction mode extends the second mode with indirect dependencies.

All three modes share the ability to filter visible edges by incoming or outgoing relations. Thereby, users can concentrate on identifying specific violations, e.g., a system library that uses modules from higher layers. Emphasizing dependency edges that cross module-borders points users at possibly forbidden relations. A measure for these cross-border dependencies is the distance between source and destination modules in the hierarchy graph when navigating through their least common ancestor. Additionally, each mode enables users to hover modules with visible dependencies and, thereby, highlight these dependencies. This way, users can distinguish them from other dependency edges. To inspect single dependency edges whose source, destination, and path may be occluded, users can highlight edges by hovering.

By using drag and drop, users can create and modify their plans to reorganize a software system's hierarchy. Fig. 3 illustrates moving a single module to its new place in a hierarchy. While moving, dependency edges follow the dragged module and the layout changes accordingly as the module leaves its old and approaches its new parent. When users choose a top-level module, its whole subtree is moved along with the module. Furthermore, selection of multiple modules enables users to move more than one module at the same time.

Creation and removal of modules enable software architects to further restructure a software's hierarchy. New modules provide a containment where users can drag submodules to. Old modules, which got empty during a restructuring process or have never actually been used, can be deleted.

When moving modules across a software hierarchy, architects may introduce new cross-border module dependencies. Fig. 4 shows how to dissolve such unwanted crossings: Users grab a dependency in its middle and drag it to a third module. Next, our refactoring tool splits up the dependency edge into two new ones, introducing an indirection, e.g., a callback mechanism in a utility library.

Another way of dissolving such cross-border dependencies is to grab them in front of a module. If the module is collapsed, dragging the dependency edge pulls out every submodule corresponding to the dependency. Otherwise, the module itself is moved similar to the drag and drop mechanism.

After finishing all virtual modifications to a software's structure, system architects get a list of intended changes, e.g., "Move module A to module B". The tool automatically removes intermediate steps and reverted actions. Using this compressed task list, architects may estimate effort requirements for planned root-canal refactorings.

IV. Implementation

We have implemented a prototype tool in C++ for visual planning of software reengineerings. Fig. 5 shows an example visualization of a software hierarchy and its dependencies using our implementation. We decided to use hierarchical edge bundles [18] for layouting. This technique uses a circular layout for the hierarchical structure and depicts edges between leaf nodes within the resulting circle.

Alternatives to hierarchical edge bundles are standard tree layout techniques such as treemaps, rooted trees or a radial layout. While treemaps provide a scalable and efficient solution to our requirements stated in Section III-B, most treemap algorithms lack stability [35]. In contrast, a rooted tree layout is stable but does not scale for large hierarchies [36]. Finally, radial layouts provide stability and better scalability than rooted trees, but similar to most hierarchical graph layouts, they suffer from visual clutter, when dependencies are drawn on top [37].

On the contrary, hierarchical edge bundles mostly satisfy our needs: First, they provide a unified view for a software system's hierarchical structure and its inner relationships. Additionally, edge bundling enables users to recognize implicitly shared parental relationships between dependencies and connected modules. Second, the circular layout is stable with respect to a hierarchy's order of modules. Hence, inserting and removing new modules into and from a hierarchy preserves the layout. Finally, hierarchical edge bundles are suitable for large-scale software systems. If necessary, lower-level modules can be collapsed and, thereby, hidden. Furthermore, edge bundling eases visual recognition of outliers. A downside of hierarchical edge bundles is their inability to visualize edges involving non-leaf nodes.

V. Case Studies

We discuss three case studies that we have performed on industrially developed and maintained software systems. First, we analyze a reengineering scenario within *BRec*, a 3D building reconstruction software by virtualcitySYSTEMS GmbH[1]. Next, we present a second case study using *SD Studio* and *SD Developer Edition*'s code base. Both software analysis and visualization tools are developed by Software Diagnostics GmbH[2]. Finally, we describe a usage scenario in a legacy COBOL software system.

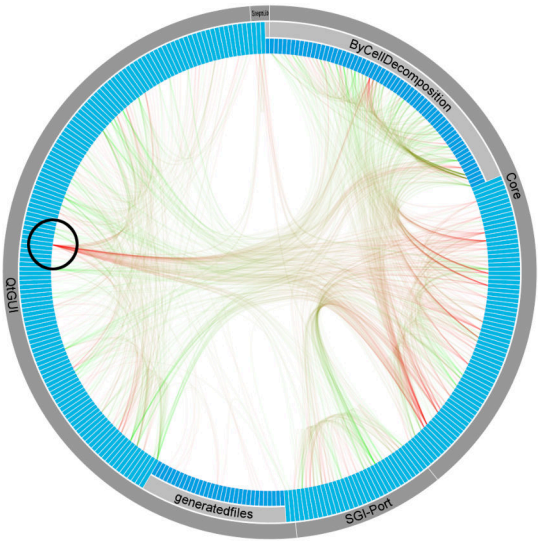


Fig. 5. Fully expanded dependency view of *BRec's* code base using hierarchical edge bundles. Nodes on outer circle represent hierarchical structure and lines depict a usage relation. Grey nodes correspond to modules, blue nodes are files. A color gradient from green to red indicates a dependency's direction.

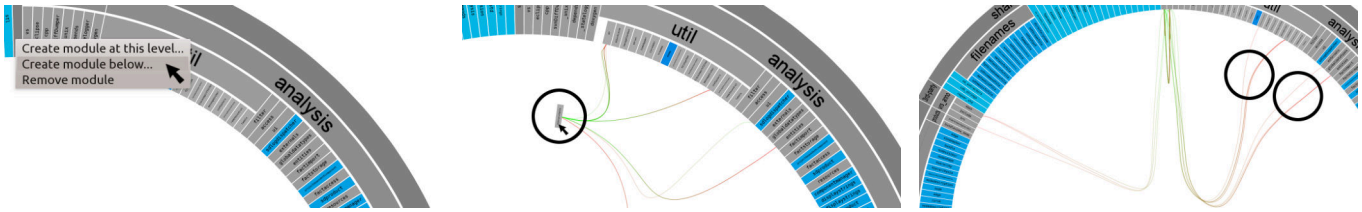
Fig. 6. Detailed view after zooming. By zooming to the file with many incoming red dependency edges, the architect found out that the *Triangle.cpp* file is violating the intended architecture.

A. virtualcitySYSTEMS: Identifying Architectural Weaknesses

BRec is a 3D building reconstruction software developed by *virtualcitySYSTEMS GmbH*. Using raw laser scan data from plane flyovers, it reconstructs and visualizes building models. It is maintained by 15 developers on average and it consists of approximately 100k LOC written in C/C++.

The task to accomplish for *virtualcitySYSTEMS'* system architect was to identify architectural weaknesses that can be quickly resolved. Using our tool, he first saw an overview of the system and its dependencies depicting approximately 350 files and 1,500 relations between them (Fig. 5). A relation between two files indicates that one file (green line end) uses one or more functions and variables defined in the other file (red line end). In this view, he noticed a large amount of

[1]http://www.virtualcitysystems.com, last accessed 5/10/2011
[2]http://www.softwarediagnostics.com, last accessed 5/10/2011



(a) (b) (c)

Fig. 7. Moving two modules to a new utility library: (a) Creation of a new utility-library subtree using a context menu. (b) Dragging the *HierarchyVis* module to its new place. Dependency edges change along with the dragged module. (c) Both modules have been moved to the utility library. New forbidden cross-module usage dependencies between the utility library and application-specific code have been introduced.

red (incoming) dependency edges connected to a single file within the *QtGUI* module. Source of these edges were files within the *Core* and *SGI-Port* modules. The architect saw this as an architectural weakness because core libraries should not depend on user interface modules. To identify the files violating this principle, he hovered over the file in the *QtGUI* module. Thereby, he highlighted the file itself, its dependent files and dependency edges in between. As nodes representing files were too small to be labeled in the overview, he zoomed in to reveal the respective file's name.

Fig. 6 shows the highlighted and zoomed *triangle.cpp* file that caused the architecture violation. The system architect wanted to estimate the amount of work required to correct this issue. He dragged the file to the *Core* module and watched the moving dependencies. He noticed no new architecture violation being introduced by the reorganization. A check of the modification list confirmed the cheapness of this restructuring in terms of required work amount.

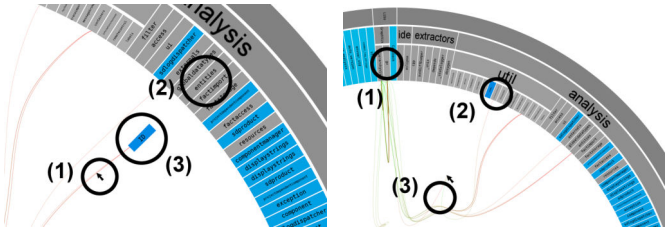
Without usage of further metrics or formal architecture descriptions, the visual software-reengineering tool enabled *virtualcitySYSTEMS'* architect to recognize an architectural violation and to estimate the effort required for correcting this issue.

B. Software Diagnostics: Decoupling Libraries

SD Developer Edition is a software tracing and visualization tool that supports developers with bug fixing and program comprehension. *SD Studio* provides visualization tools, such as software maps, to assess a software system's quality, source code evolution and development resources. Both tools share a code base that comprises approximately 230k LOC within 2,100 C++ files.

To improve reusability, Software Diagnostics' system architect planned to remove two graph visualization submodules from the application code-base and to put them into a new "graphvis" utility library. This included eliminating all dependencies from both modules to application-specific code. Based on experiences of *Software Diagnostic's* developers, the architect expected forbidden dependencies. To avoid unknown risks, he decided to make an effort estimation prior to assigning the task to the developers.

Using our prototypical tool, the system architect visually explored the steps required to fulfill the reengineering. To



(a) (b)

Fig. 8. Dissolving unwanted dependencies: (a) By grabbing the dependency edge (1) in front of the *Entities* module (2), the architect pulled out the *ID* module (3) that caused the unwanted dependency. (b) *Software Diagnostics'* system architect grabbed the dependency edge between the *GV* module (1) and the *Userfeedback* module (2) and, thereby, separated it into two new dependencies (3). Afterwards, he could put these new dependencies into the *Callback* module.

concentrate on high-level modules, he first collapsed low-level layers. Fig. 7 illustrates his next three steps: First, he created a new container module for the two libraries to move (Fig. 7(a)). He dragged both libraries towards the container module and dropped them (Fig. 7(b)). During dragging, he already recognized several dependency edges indicating usage of application-specific code within both graph visualization modules: They used code from the application-specific modules *Entities*, *Containers*, *Userfeedback* and *Layouting* (Fig. 7(c)).

Relying on his knowledge of the affected modules, the architect used different strategies to dissolve these dependencies. For the *Containers* module, he decided to place it into the new container module as well, because it consists of low-level re-usable collection classes. To find out, which part of the *Entities* module was actually referenced by the graph visualization modules, the architect grabbed the dependency edge in front of the module and pulled out a module *ID* (Fig. 8(a)). As the *ID* module is low-level functionality as well, he moved it to the new utility library. To decouple the graph visualization libraries and the *Userfeedback* module, which is responsible for displaying progress dialogs to users, he created a new *Callback* utility library and grabbed the corresponding dependency edge in its middle. By dragging the grabbed edge to the *Callback* module, he divided the dependency into two new ones: Both, the graph visualization modules and the *Userfeedback* module used the new *Callback* library (Fig. 8(b)). To remove the last cross-border dependency to the *Layouting* module, the architect decided to put it directly

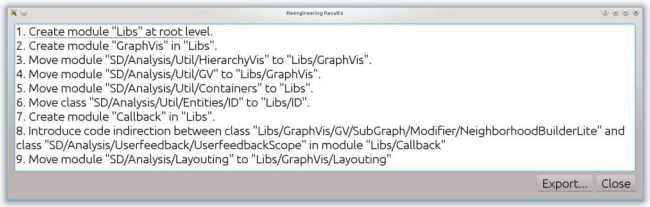


Fig. 9. Dialog displaying a detailed task list for decoupling two modules from *Software Diagnostics'* application-specific code base. The list was generated based on the system architect's interactions with our prototypical tool.

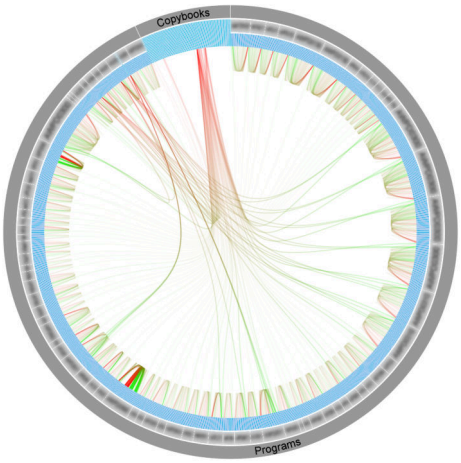


Fig. 10. Dependency view of an industrial COBOL system. Grey nodes represent programs and copybooks, blue nodes are sections within programs. Edges correspond to dependencies such as *PERFORM*, *CALL*, or *COPY* statements. Due to intellectual property reasons, we had to obfuscate most entity labels.

into the new graph visualization library as layouting is a central task for graph visualization.

Having eliminated each forbidden dependency, Software Diagnostic's system architect retrieved a list of modifications from the tool (Fig. 9). Using this list, he estimated the effort required to fulfill the decoupling task and eventually decided to assign it to developers.

The visualization technique helped to break down a generic reengineering task, i.e., decoupling of two graph visualization libraries from the application code-base, into clearly formulated subtasks without reading or changing any code. Risks such as unexpected dependencies and architecture violations could be estimated beforehand.

C. COBOL: Change Impact in Legacy Software Systems

The subject software system of this case study is an excerpt of a confidential 100k LOC COBOL system from the banking industry (Fig. 10). It consists of approximately 1,100 sections in 150 programs and copybooks (include files). On the uppermost structure level, the visualization separates a copybook and a program section. Next, actual programs and copybooks follow. Finally, programs are further separated into sections based on their procedure division, i.e., a COBOL program's behavioral part.

Besides introducing a new developer to the code base using a visual representation of its structure, our prototypical tool has been used for change impact analyses. By hovering programs to be modified, the system's maintainer visually explored dependent sections and programs. They might be affected because they are called by a subject program or, vice versa, it is called by them. A similar task was to find all programs potentially affected by a copybook modification.

Our tool enabled the maintainer to efficiently estimate efforts required for an intended modification by analyzing its impact visually. He identified program sections that required changes as well and are subject to subsequent quality assurance. Thereby, the risk of introducing new bugs was reduced.

VI. Limitations

During our case studies, we identified some limitations of our concept. First, planning fine-grained reengineerings such as interface changes requires detailed system knowledge, which is hard to obtain using the proposed concept. Additionally, some interactions, e.g., introducing an indirection into a dependency, can lead to unpredictable code modifications within affected modules. Hence, our tool's resulting subtask list may be too coarse-grained in such cases so that architects need additional knowledge to determine reasonable effort estimations. In contrast, a reengineering's result list may become quite large and it can thus be hard for software architects to overview the entire list. In this case, its textual representation aggravates a sensible effort estimation. Introducing a detailed code model and linking the visualization to synchronized code views could provide additional support here, but would as well reduce generalizability of our concept.

Furthermore, our visualization concept does not support differentiating between distinct dependency types. Call relations, for instance, cannot be distinguished from type usages. In addition, defining a single generic modular view of a software system's structure limits our concept's applicability to analysis of one hierarchy at a time. A possible solution, though, is to depict one view per hierarchy (or dependency type) and synchronize these upon modifications.

VII. Conclusions

Software redesigns and reengineerings represent crucial processes in software maintenance. Improving comprehensibility and maintainability of aged and, in particular, large systems saves major expenses. This compensates the risk of breaking existing functionality. Nevertheless, experience from industry shows that in particular complex and extensive reengineerings are typically deferred or even avoided at all. One reason for this is the difficulty in estimating feasibility and required effort beforehand using state-of-the-art tools.

We presented a concept enabling software architects to visually analyze and design changes to software systems. Using a unified view of a software system's hierarchy and inner dependencies, software architects can reorganize a software's modular structure interactively. Drag and drop, a dependency grabbing metaphor, and virtual creation and removal

of software entities enable architects to experiment with a software's design and test their hypotheses. A list of performed modifications assists architects with their effort estimation.

We applied this concept to a prototypical implementation using hierarchical edge bundles and tested it with three large-scale software systems. These case studies showed that software architects can verify assumptions concerning code structure and plan reengineerings using our tool. Thereby, they achieved an improvement of their effort estimations.

As future work, we plan to extend our prototypical implementation to support projecting aggregated code quality metrics onto a software's hierarchy representation. This enables architects to identify potential cost-intensive design flaws, e.g., modules that are frequently modified and contain forbidden dependencies. Moreover, zooming capabilities down to class member and function level would further improve a software architect's means to dissolve dependencies. For example, for a class violating the separation of concerns principle, architects could disentangle dependencies on member level.

Further future work includes enhancing hierarchical edge bundles to support dependencies between non-leaf nodes. Performing controlled experiments provides means to trace software architects' planning workflows and to compare user satisfaction with different layout algorithms and interaction techniques.

Acknowledgements

We would like to thank *Software Diagnostics GmbH* and virtualcitySYSTEMS GmbH for providing their code bases and performing the case studies with us. This work was supported by the ZIM program of the Federal Goverment of Germany (BMWi).

References

[1] M. Fowler, *Refactoring - Improving the Design of Existing Code*. Addison-Wesley Professional, 2000.

[2] E. J. Chikofsky and J. H. Cross II, "Reverse engineering and design recovery: A taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13–17, 1990.

[3] T. M. Pigoski and A. April, *Software Engineering Body of Knowledge*. IEEE Computer Society, 2004, ch. Software Maintenance, pp. 6.1–6.16.

[4] K. Beck, *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 1999.

[5] M. Cohn, *Succeeding with Agile: Software Development Using Scrum*, K. Gettman, Ed. Addison-Wesley, 2009.

[6] E. Murphy-Hill and A. P. Black, "Refactoring tools: Fitness for purpose," *IEEE Software*, vol. 25, pp. 38–44, Sep. 2008.

[7] G. C. Murphy, M. Kersten, and L. Findlater, "How are java software developers using the eclipse ide?" *IEEE Software*, vol. 23, pp. 76–83, Jul. 2006.

[8] P. Weissgerber and S. Diehl, "Are refactorings less error-prone than other changes?" in *Proceedings of the International Workshop on Mining Software Repositories*, 2006, pp. 112–118.

[9] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *IEEE Transactions on Software Engineering*, vol. 29, no. 3, pp. 210–224, 2003.

[10] D. L. Parnas, "Software aging," in *International Conference on Software Engineering*, 1994, pp. 279–287.

[11] A. Telea and L. Voinea, "Case study: Visual analytics in software product assessments," in *Proceedings of the International Workshop on Visualizing Software for Understanding and Analysis*, 2009, pp. 65–72.

[12] F. Bourquin and R. K. Keller, "High-impact refactoring based on architecture violations," in *Proceedings of the European Conference on Software Maintenance and Reengineering*, 2007, pp. 149–158.

[13] S. Demeyer, S. Ducasse, and O. Nierstrasz, *Object Oriented Reengineering Patterns*. Morgan Kaufmann Publishers Inc., 2002.

[14] F. Simon, F. Steinbrückner, and C. Lewerentz, "Metrics based refactoring," in *Proceedings of the European Conference on Software Maintenance and Reengineering*, 2001, pp. 30–38.

[15] S. Roock and M. Lippert, *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. John Wiley & Sons, Inc., 2006.

[16] M. Lanza and S. Ducasse, "Polymetric views-a lightweight visual approach to reverse engineering," *IEEE Transactions on Software Engineering*, vol. 29, no. 9, pp. 782–795, 2003.

[17] R. Wettel and M. Lanza, "Visually localizing design problems with disharmony maps," in *Proceedings of the Symposium on Software visualization*, 2008, pp. 155–164.

[18] D. Holten, "Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, pp. 741–748, 2006.

[19] J. Bohnet and J. Döllner, "Monitoring code quality and development activity by software maps," in *Proceedings of the International Workshop on Managing Technical Debt*, 2011.

[20] C. Lewerentz, F. Simon, and F. Steinbrückner, "Crococosmos," in *Graph Drawing*, ser. Lecture Notes in Computer Science. Springer, 2002, vol. 2265, pp. 72–76.

[21] R. W. Schwanke, "An intelligent tool for re-engineering software modularity," in *Proceedings of the international conference on Software engineering*, 1991, pp. 83–92.

[22] P. Bengtsson and J. Bosch, "Scenario-based software architecture reengineering," in *Proceedings of the International Conference on Software Reuse*, 1998, pp. 308–317.

[23] B.-K. Kang and J. M. Bieman, "Using design abstractions to visualize, quantify, and restructure software," *Journal of Systems and Software*, vol. 42, pp. 175–187, Aug. 1998.

[24] R. Marinescu, "Detecting design flaws via metrics in object-oriented systems," in *Proceedings of the International Conference and Exhibition on Technology of Object-Oriented Languages and Systems*, 2001, pp. 173–182.

[25] D. Ratiu, S. Ducasse, T. Gîrba, and R. Marinescu, "Using history information to improve design flaws detection," in *Proceedings of the Euromicro Working Conference on Software Maintenance and Reengineering*, 2004, pp. 223–232.

[26] R. W. Schwanke and S. J. Hanson, "Using neural networks to modularize software," *Machine Learning*, vol. 15, pp. 137–168, May 1994.

[27] D. H. Hutchens and V. R. Basili, "System structure analysis: Clustering with data bindings," *IEEE Transaction on Software Engineering*, vol. 11, pp. 749–757, Aug. 1985.

[28] S. C. Choi and W. Scacchi, "Extracting and restructuring the design of large systems," *IEEE Software*, vol. 7, pp. 66–71, Jan. 1990.

[29] E. Murphy-Hill and A. P. Black, "An interactive ambient visualization for code smells," in *Proceedings of the International Symposium on Software Visualization*, 2010, pp. 5–14.

[30] H. A. Müller and K. Klashinsky, "Rigi—a system for programming-in-the-large," in *Proceedings of the International Conference on Software Engineering*, 1988, pp. 80–86.

[31] P. Mayer, A. Meissner, and F. Steimann, "A visual interface for type-related refactorings," in *Proceedings of the Workshop on Refactoring Tools*, 2007, pp. 5–6.

[32] N. Tsantalis and A. Chatzigeorgiou, "Identification of refactoring opportunities introducing polymorphism," *Journal of Systems and Software*, vol. 83, pp. 391–404, Mar. 2010.

[33] S. Hayashi, M. Saeki, and M. Kurihara, "Supporting refactoring activities using histories of program modification," *IEICE Transactions on Information and Systems*, vol. E89-D, pp. 1403–1412, Apr. 2006.

[34] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Addison-Wesley, 2007.

[35] B. Shneiderman and M. Wattenberg, "Ordered treemap layouts," in *Proceedings of the IEEE Symposium on Information Visualization*, 2001, pp. 73–78.

[36] I. Herman, G. Melançon, and M. S. Marshall, "Graph visualization and navigation in information visualization: A survey," *IEEE Transactions on Visualization and Computer Graphics*, vol. 6, no. 1, pp. 24–43, 2000.

[37] R. A. Becker, S. G. Eick, and A. R. Wilks, "Visualizing network data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 1, pp. 16–28, 1995.